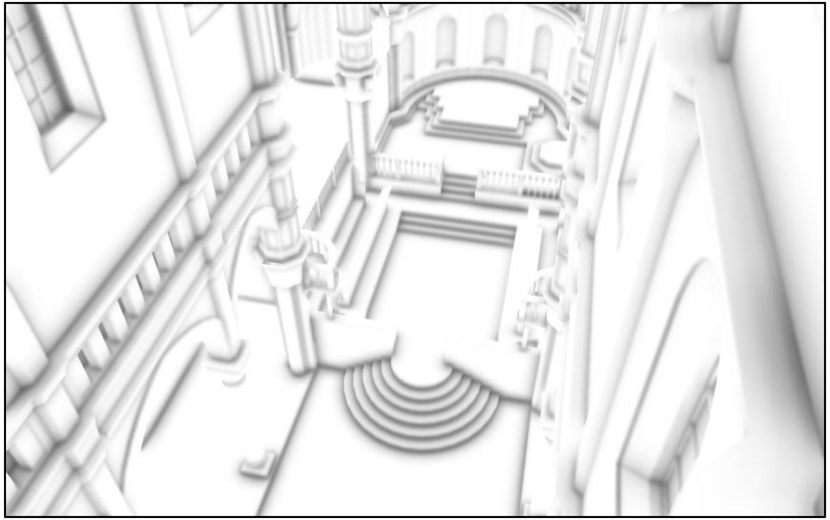
Particle Shadows & Cache-Efficient Post-Processing

Louis Bavoil & Jon Jansen
Developer Technology, NVIDIA

GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 25–29, 2013
EXPO DATES: MARCH 27–29
2013

Agenda

1. Particle Shadows



2. Cache-Efficient Post-Processing

Part 1:
Particle Shadows

Particle Shadows

Assumption

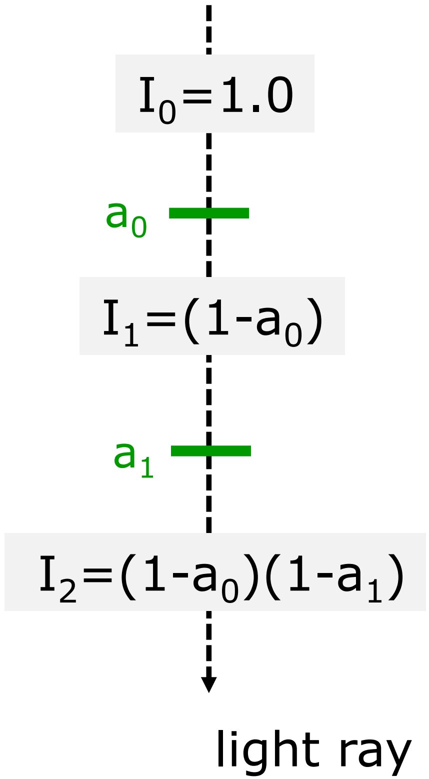
Each particle transmits (1-alpha) of its incoming light intensity

Definition

Shadow cast by particles along a given light-ray segment

= Transmittance

= $(1-a_0)(1-a_1) \ldots (1-a_{N-1})$



"External Shadows"

Idea

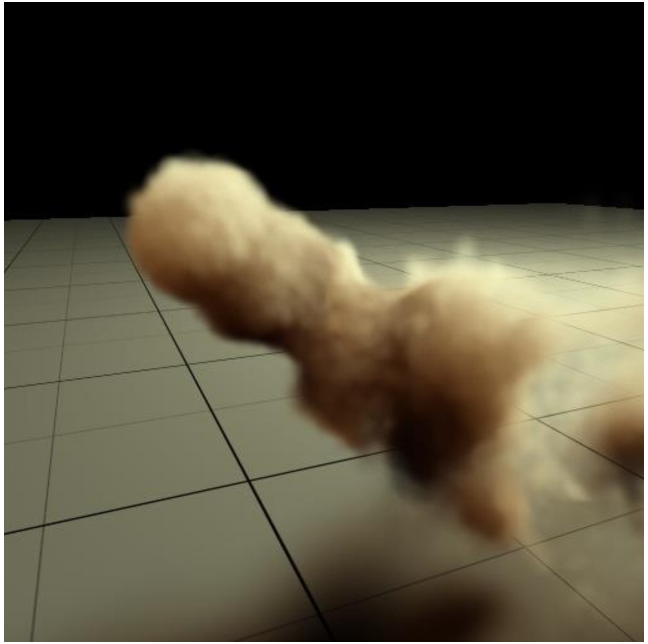
Blend $(1-a_0)(1-a_1) \ldots (1-a_{N-1})$ to a R8_UNORM "Translucency Map" [Crytek 2011]

Pros

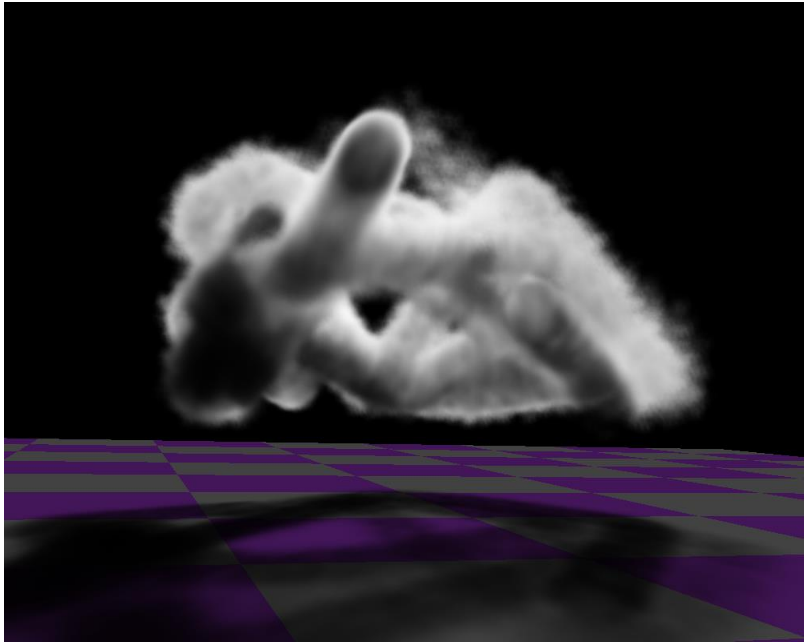
1. Compact memory footprint
2. Map rendered in one pass, order-independent
3. Fast shadow projection: R8_UNORM bilinear fetch

Screenshot from [Crytek 2011]

Wanted: Particle Self-Shadows



[Green 2012]



[Jansen 2010]

Volumetric Self-Shadowing

Large body of research work

- Deep Shadow Maps [Lokovic 2000]
- Opacity Shadow Maps [Kim 2001] [NVIDIA 2005]
- Deep Opacity Maps [Yuksel 2008]
- Adaptive Volumetric Shadow Maps [Salvi 2010]
- **Fourier Opacity Mapping (FOM)** [Jansen 2010] (*)
- Extinction Transmittance Maps [Gautron 2011]
- Half-Angle Slicing [Green 2012] [Kniss 2003]

(*) Shipped in “Batman: Arkham Asylum” (PC)

Wanted: Scalability

- Build on shadow mapping
  - Extend existing opaque-shadow systems
  - Support large scenes, multiple lights
- Support large shadow depth ranges
  - Do not get limited by MRTs

Wanted: Lots of Detail

Goal: reveal structural detail

Our Solution:

Particle Shadow Mapping

"Particle Shadow Map"

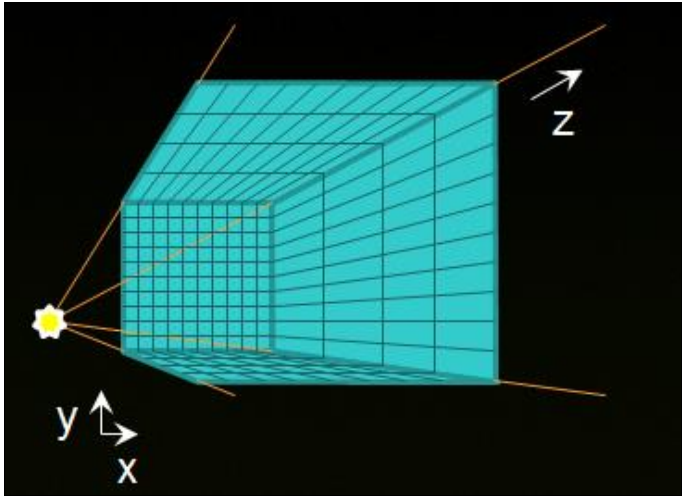
PSM = 3D Texture

Mapped into light space

xy/uv planes are always perpendicular to light rays

Store shadow per voxel

(transmittance through light ray up to that voxel)



PSM Algorithm

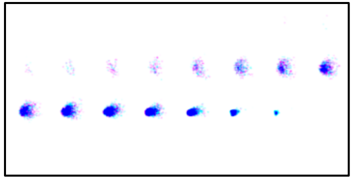
STEP 1: Clear PSM to 1.f everywhere

STEP 2: Voxelize particle transmittances to PSM

STEP 3: Propagate transmittances along rays through PSM

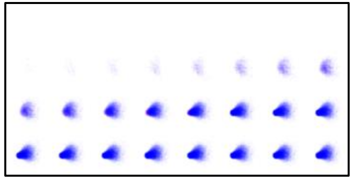
STEP 4: Sample transmittance from PSM when rendering scene

STEP 1



STEP 2

[VS+GS+PS+Blend]



STEP 3

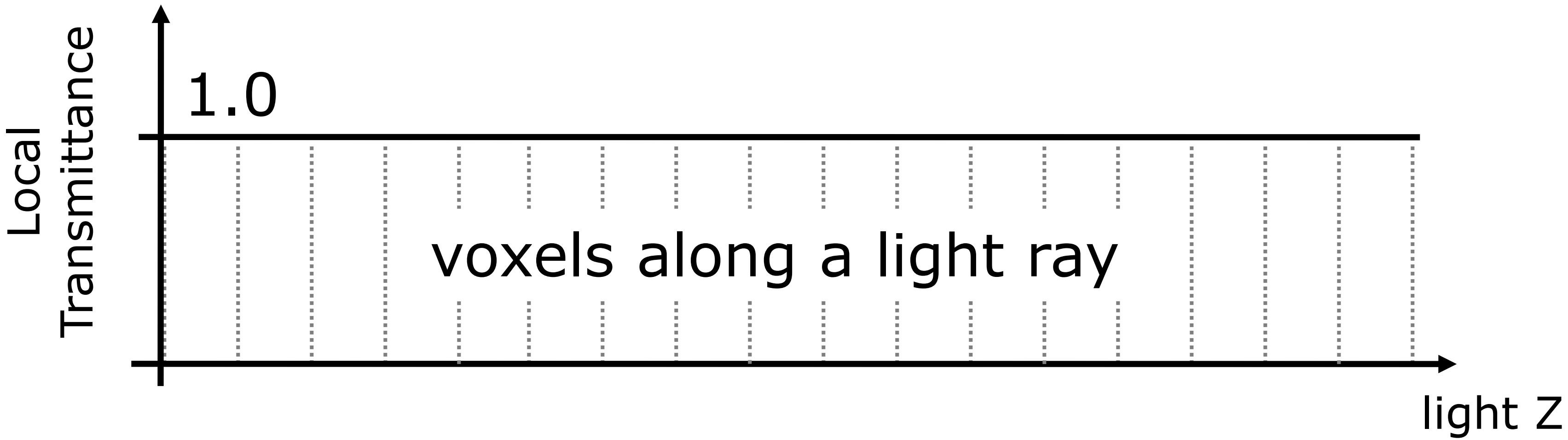
[CS]

STEP 4

PSM Layout

3D Texture representing voxelized local transmittances

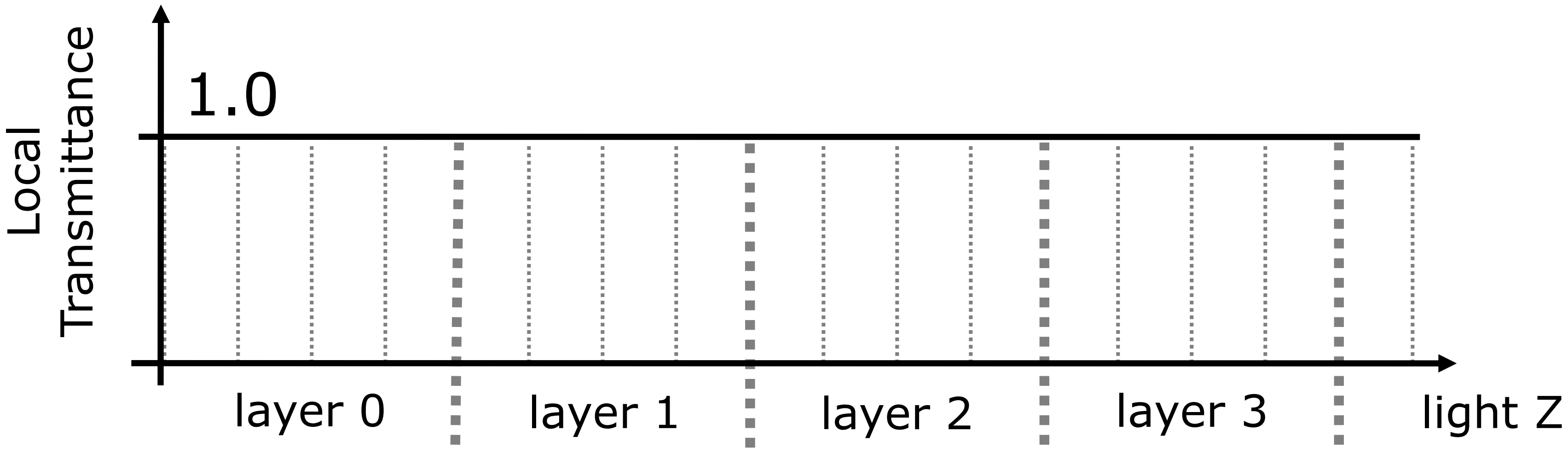
Storing FP32 transmittances would be overkill



PSM Layout

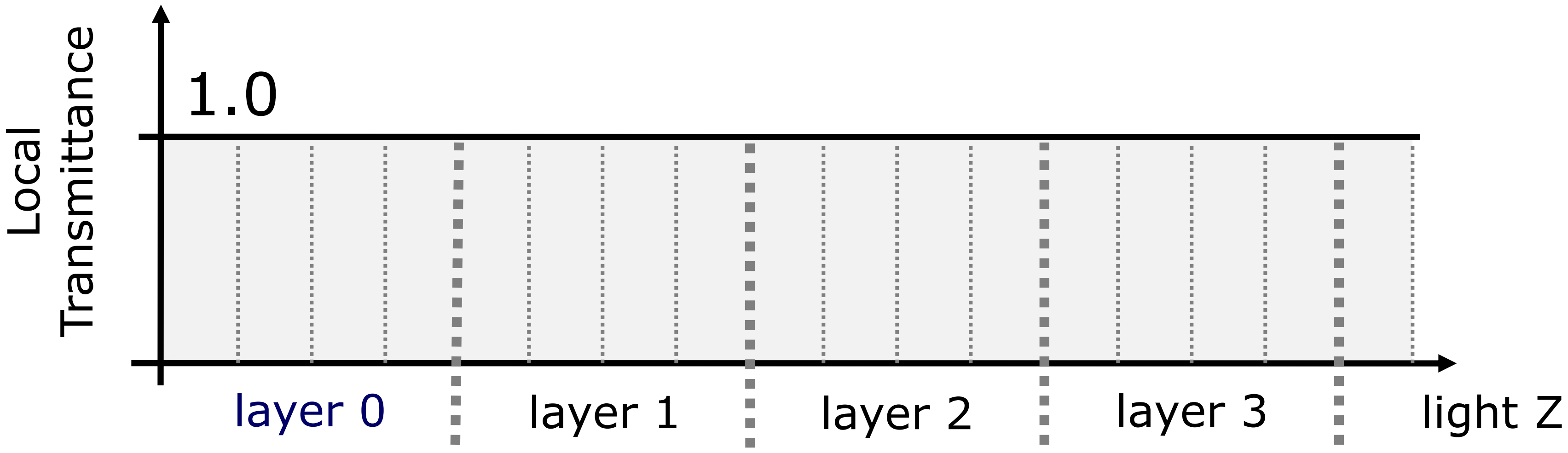
Can pack 4 x 8-bit values into one 4x8_UNORM

e.g. 256^3 PSM stored as 256x256x64 4x8_UNORM texture

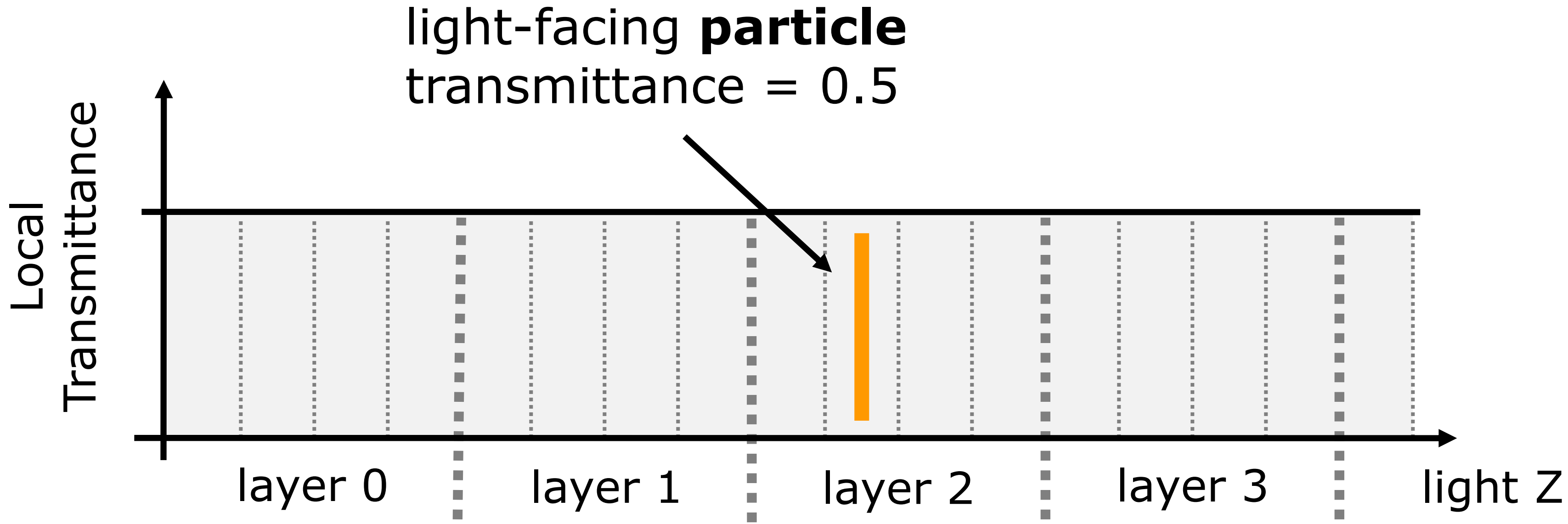


Step 1: Clear PSM

Clear 3D Texture to 1.0 (no shadow)

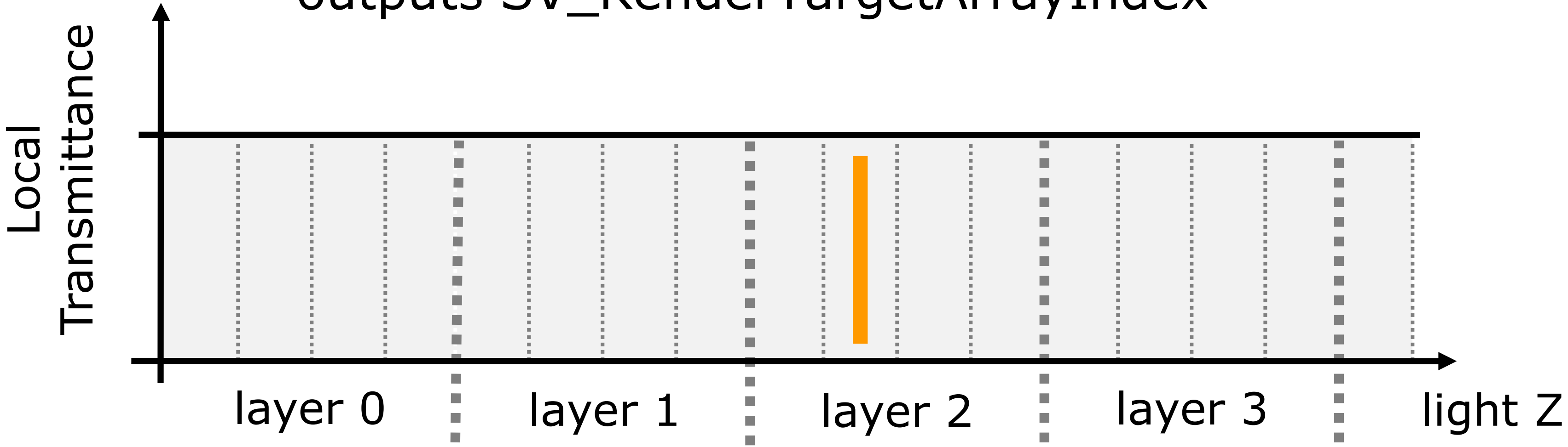


Step 2: Voxelize Transmittances



Step 2: Voxelize Transmittances

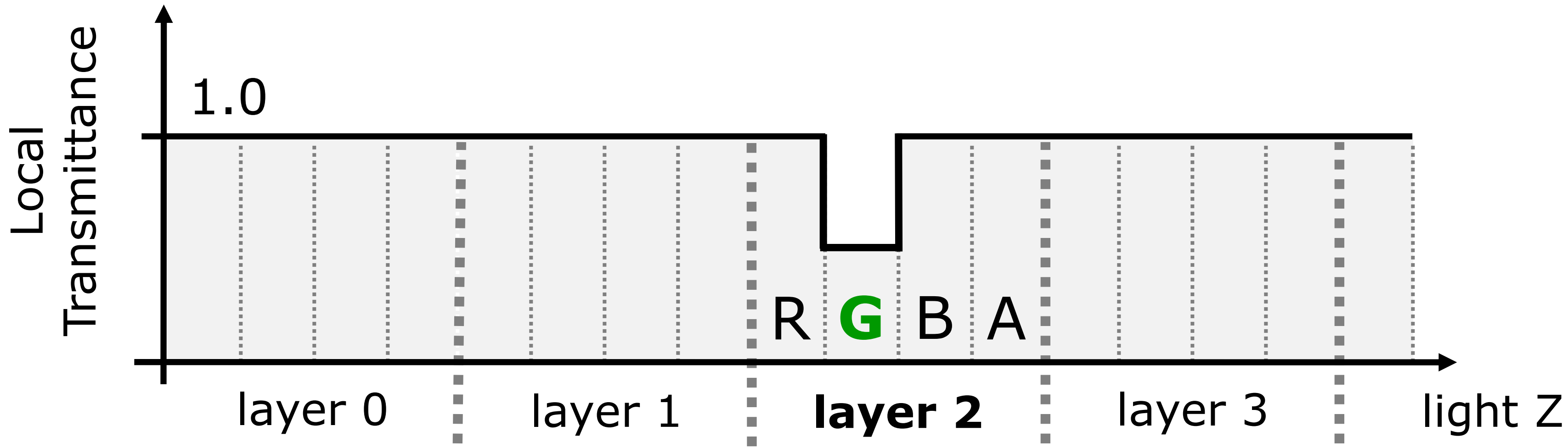
Geometry Shader
with [maxvertexcount(4)]
outputs SV_RenderTargetArrayIndex *



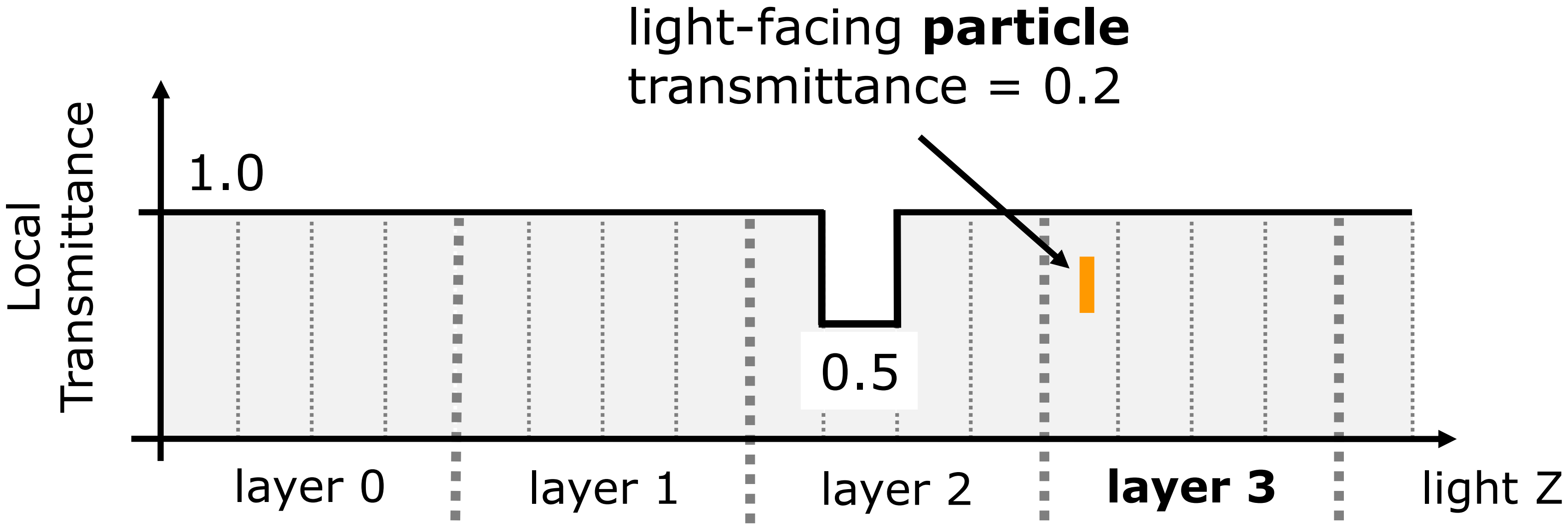
* Works because shadow casters are particles. Hence the name "**Particle** Shadow Mapping".

Step 2: Voxelize Transmittances

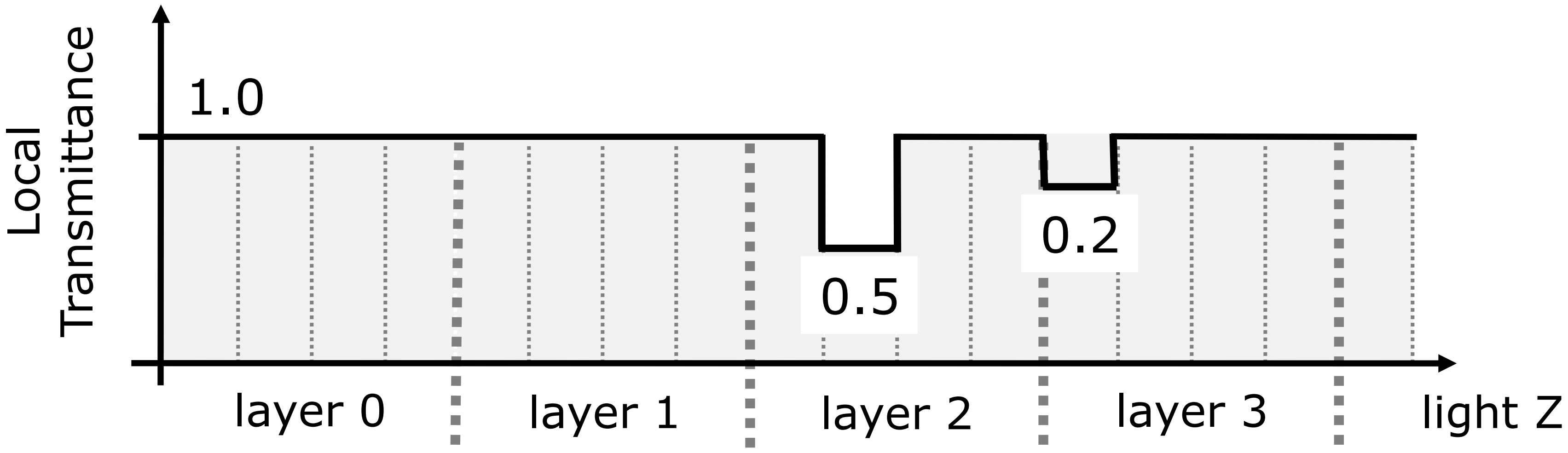
GS assigns particle to layer=2, channel=G
PS writes (1.f-alpha) to G, and 1.f to R,B,A
OM does **Multiplicative Blending**



Step 2: Voxelize Transmittances

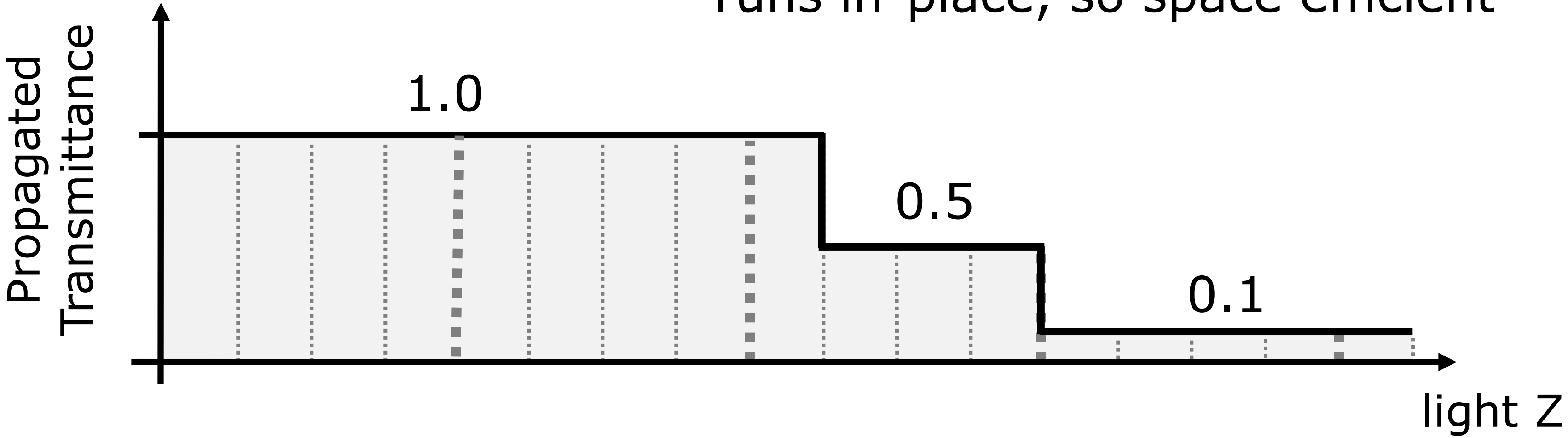


Step 2: Voxelize Transmittances



Step 3: Propagate Transmittances

Compute Shader
with one thread per light ray
runs in-place, so space efficient



Step 4: Sample from PSM

Output from STEP 3

= Particle Shadow Map

= Per-Voxel Shadows

Shadow Evaluation

Cannot use a trilinear texture fetch due to RGBA packing

So perform 2 bilinear fetches & lerp between slices

PSM Practicality

Obvious objection to PSM is space complexity e.g.

- 256x256x256 x 8bits = 16MB (= 0.78% of 2GB FB)
- 512x512x512 x 8bits = 128MB (= 6.25% of 2GB FB)

Arguably

- 256^3 is feasible right now
- 512^2 x 256 (= 64MB) could work as ‘extreme’ setting

Comparison to External Shadows

	External Shadows [Crytek 2011]	**PSM**
Render shadow map	RT=1x8bits	RT=1x32bits
Propagation	n/a	O(w x h x d)
Sample shadow map	1 texture lookup/sample	2 texture lookups/sample
Space complexity	O(w x h)	O(w x h x d)

Comparison to Prior Art

	MRT OSM [NVIDIA 2005]	**Half-Angle Slicing** [Green 2012]	**FOM** [Jansen 2010]	**PSM**
Render to shadow map	MRT=dx8bits	MRT=1x8bits	MRT=dx16bits	MRT=1x32bits
Render to shadow map RT changes	1	O(d)	1	1
Propagation	n/a	n/a	n/a	O(w x h x d)
Sample shadow map textures	O(d) fetches	1 fetches	O(d) fetches	2 fetches
Space complexity	O(w x h x d)	O(w x h)	O(w x h x d)	O(w x h x d)

PSM Performance

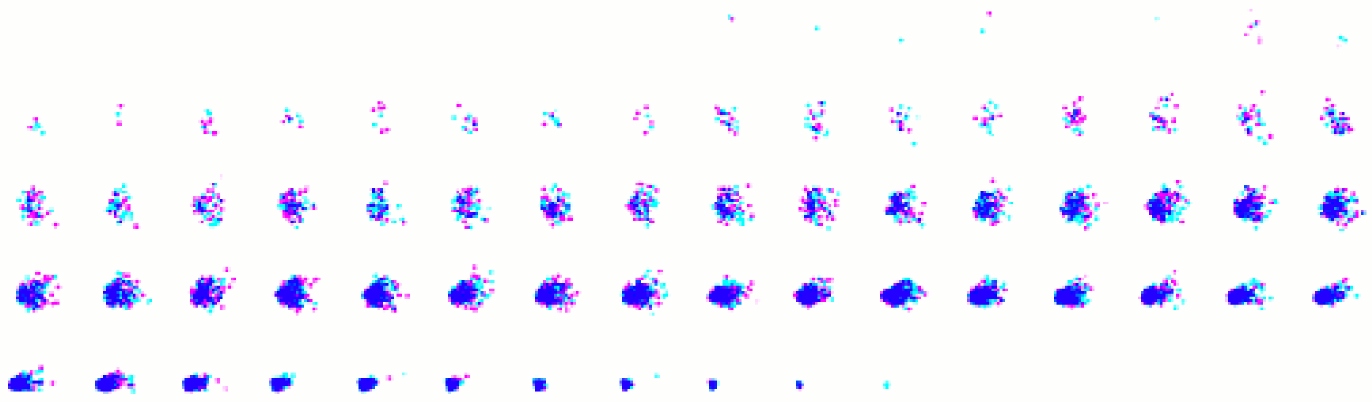
8K large particles

256^3 Particle Shadow Map

PSM Generation	GPU Time *
PSM RT clear	0.01 ms
Render to PSM	0.23 ms
Propagation CS	0.33 ms
Total	**0.58 ms**

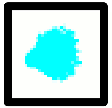
* Measured with D3D11 timestamp queries on GTX 680

Output of STEP 2:
Voxelized Local Transmittances

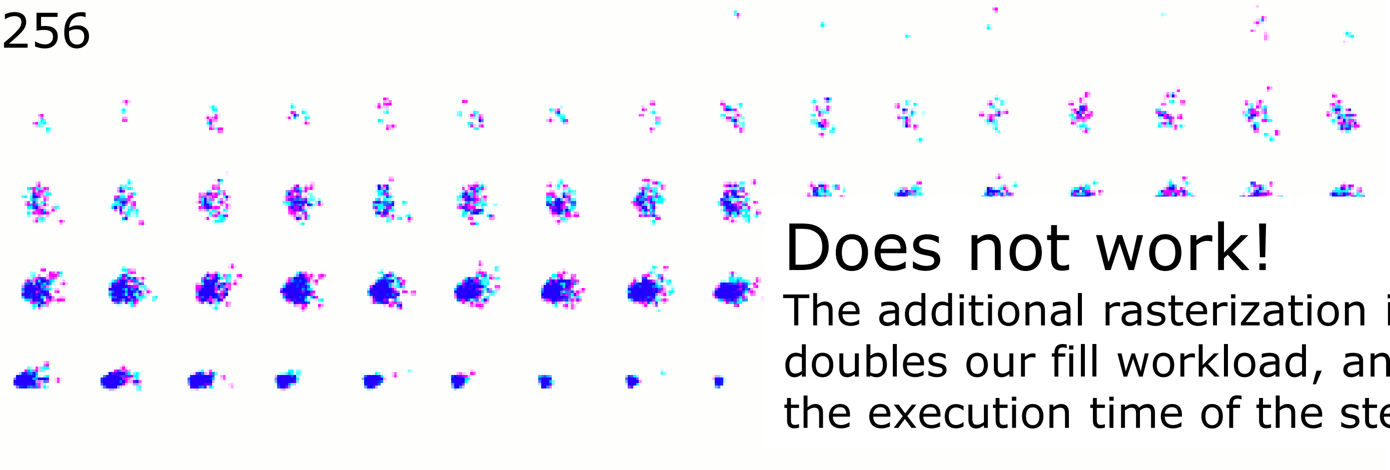


Coverage Optimization

Goal: in STEP 3, early exit for "empty light rays"

256  ← IDEA 1: slice 0 reserved for coverage

256



Does not work!

The additional rasterization into slice 0 doubles our fill workload, and therefore the execution time of the step

Coverage Optimization

Solution: Output particles to 2 D3D11 viewports

GS output #0 → (Layer 0, Viewport 0)
conservative coverage mask
[8x8 resolution]

GS output #1 → (Layer >0, Viewport 1)
entire PSM slice, as before
[256^2 resolution]

Coverage Optimization

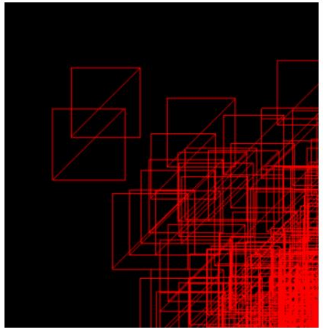
PSM Generation	No Opt	Opt	Speedup
PSM RT clear	0.01 ms	0.01 ms	0%
Render to PSM	0.23 ms	0.26 ms	-11%
Propagation CS	0.33 ms	0.23 ms	43%
Total	**0.58 ms**	**0.50 ms**	**16%**

256^3 PSM, 8K large particles, GTX 680 timings

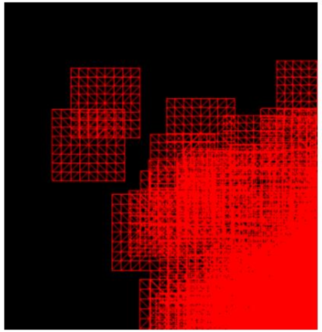
Particle Lighting with DX11

When rendering particles to scene color buffer

- Can render particles with DX11 tessellation
- And fetch shadow maps in DS instead (faster than PS)



un-tessellated



tessellated

See Bitsquid's GDC'12 talk on "Practical Particle Lighting"
[Persson 2012]

And NVIDIA's "Opacity Mapping" DX11 Sample
[Jansen 2011]

PSM Wrap Up

"Particle Shadow Mapping" (PSM)

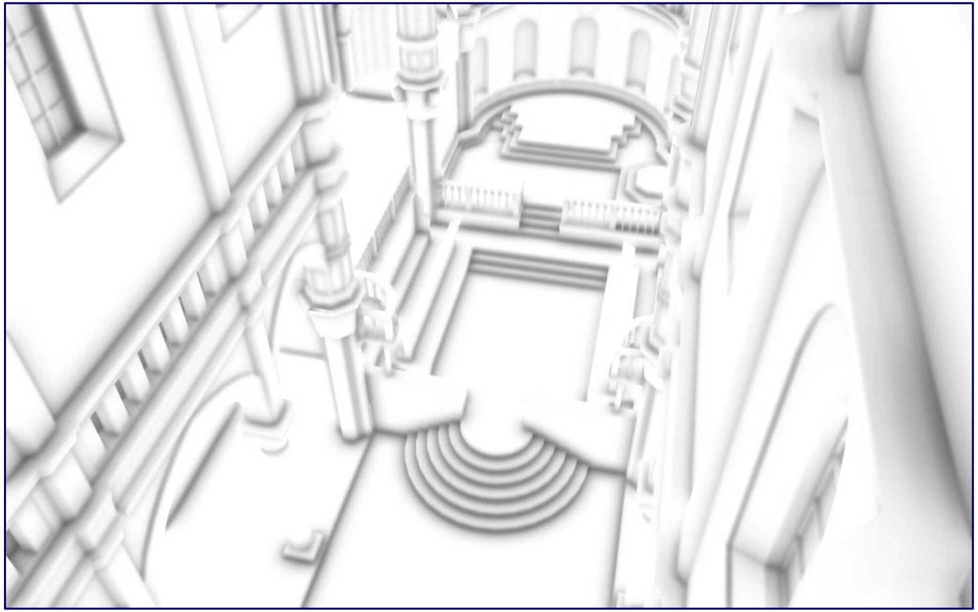
- Specialized OSM technique for particles shadows
- Scattering particles to 3D-texture slices

D3D11 features used

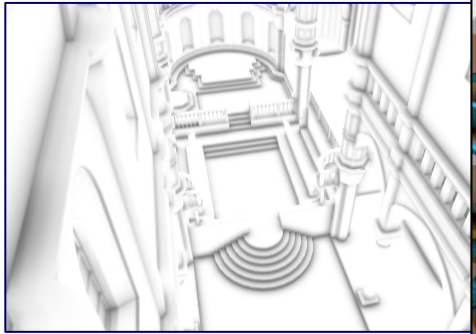
- **GS** for particle expansion + voxelization + coverage opt
- **CS** for transmittance propagation
- **DS** for fetching the PSM faster than in PS

DEMO

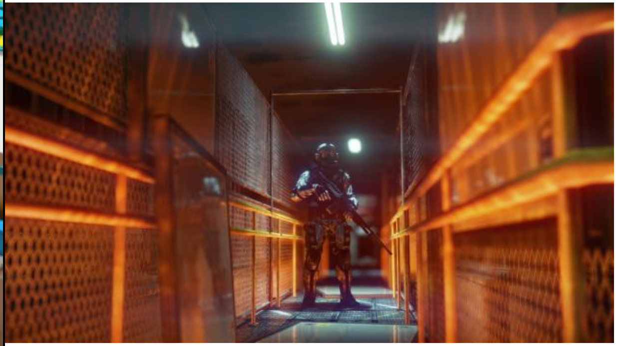
Part 2:
Cache-Efficient Post-Processing



Large, Sparse & Jittered Filters


SSAO

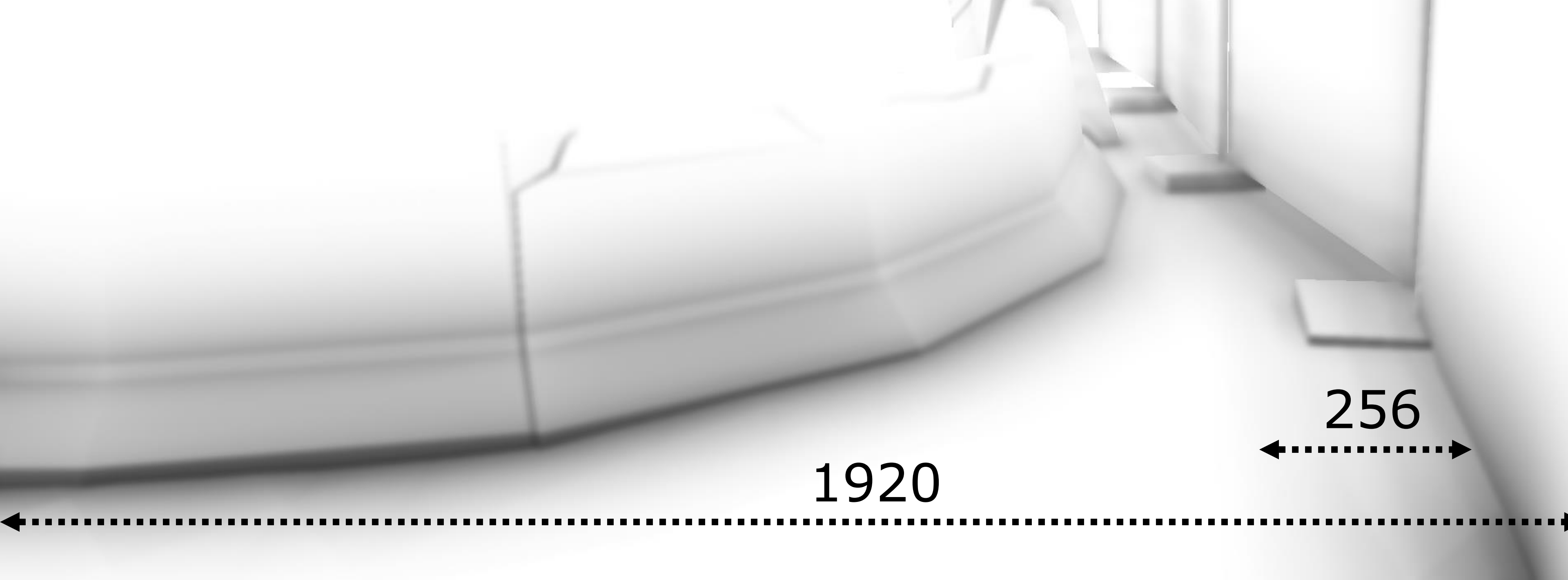
SSDO [Ritschel 2009]


SSR [Crytek 2011]

Goal: Generic approach to speedup such filters without sacrificing quality

Large, Sparse & Jittered Filters

Kernel size up to 512x512 texels



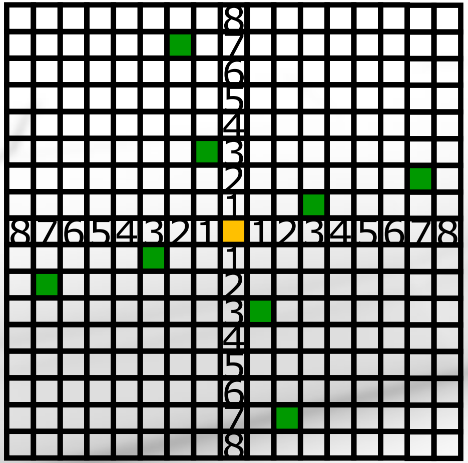
Large, **Sparse** & Jittered Filters

e.g. 8 samples in 256^2 area

Difficult to accelerate with a Compute Shader

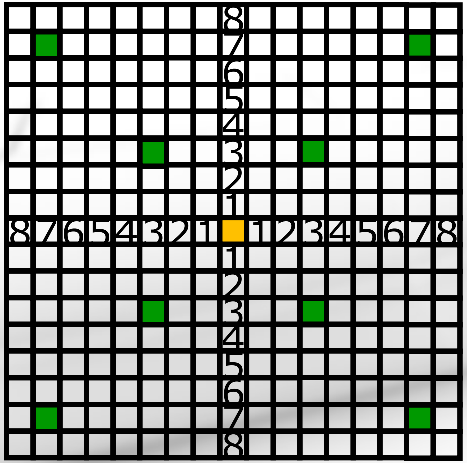
Large, Sparse & **Jittered** Filters

Adjacent pixels have different sampling patterns



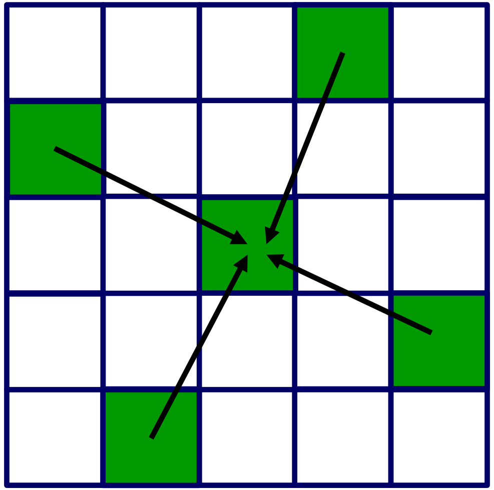
Large, Sparse & **Jittered** Filters

Adjacent pixels have different sampling patterns



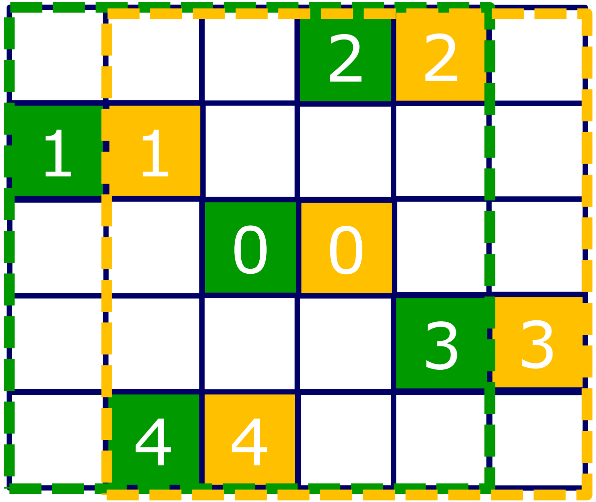
Fixed Sampling Pattern

Example kernel



Fixed Sampling Pattern

Now, for a pair of adjacent pixels executed in lock step

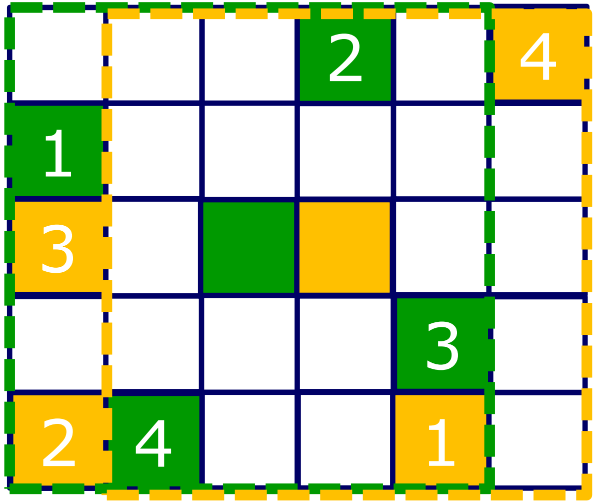


For each sample,
adjacent pixels fetching
adjacent texels

➔ Good spatial locality ☺

Random Sampling Pattern

Randomizing the texture coordinates per pixel…

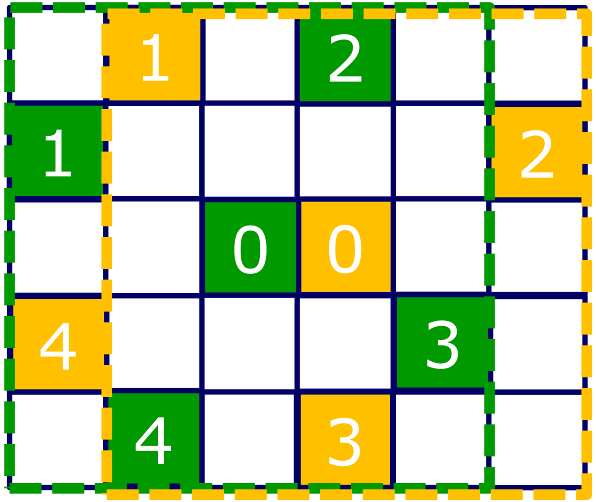


For each sample,
adjacent pixels fetching
far-apart texels

➔ Poor spatial locality ☹

Jittered Sampling Pattern

Jitter each of the 4 samples within 1/4th of kernel area



For each sample, **adjacent pixels** fetching **sectored texels**

➔ Better spatial locality

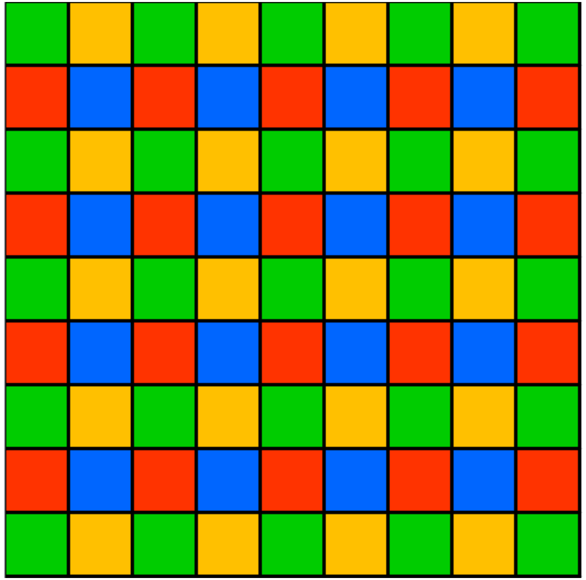
… but as kernel size increases, sector size increases too ☹

Previous Art

1. Jittered sampling patterns
   Jitter within one sector
2. Mixed-resolution inputs
   Use full-res texture for center tap
   Use low-res texture for sparse samples
3. MIP-mapped inputs [McGuire 2012]

Still, remaining **per-pixel jittering** hurts **per-sample locality**

Assumption:
Interleaved Sampling Patterns

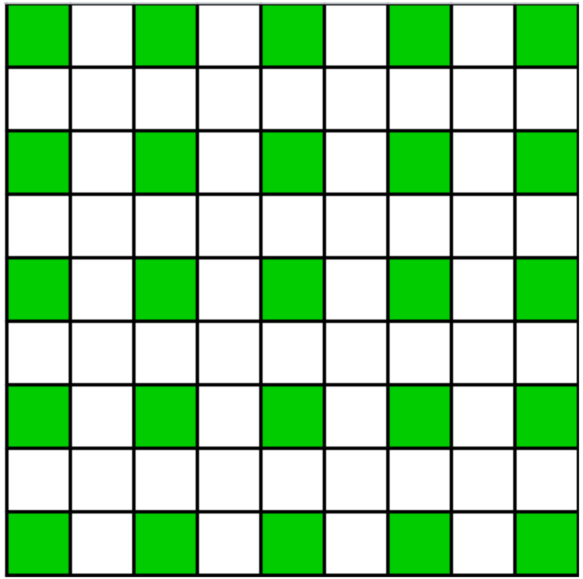


NxN sampling patterns interleaved on screen

Typical sampling strategy for SSAO, SSDO, SSR, etc.

Per-pixel jitter seed fetched from a tiled “jitter texture”

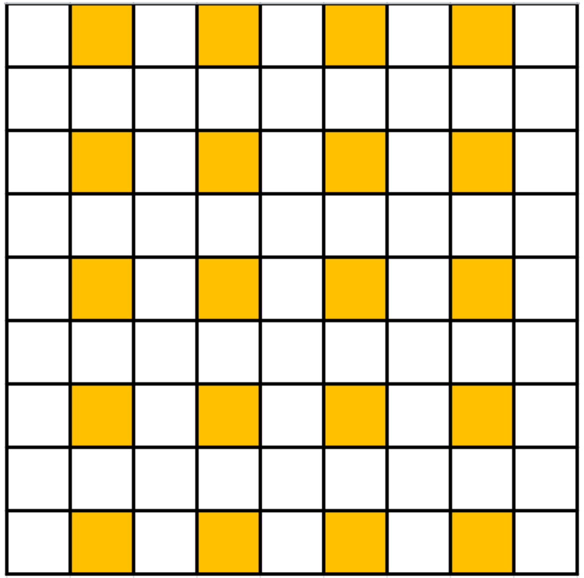
Approach



“**individually render lower resolution images corresponding to the regular grids**, and to then interleave the samples obtained this way by hand”

[Keller 2001]

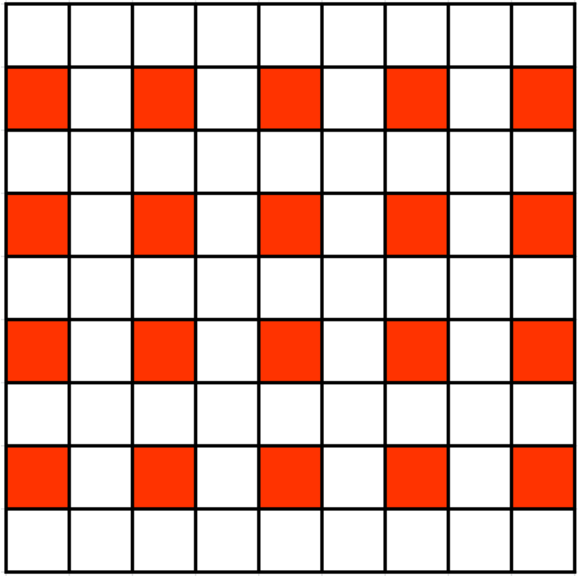
Approach



“**individually render lower resolution images corresponding to the regular grids**, and to then interleave the samples obtained this way by hand”

[Keller 2001]

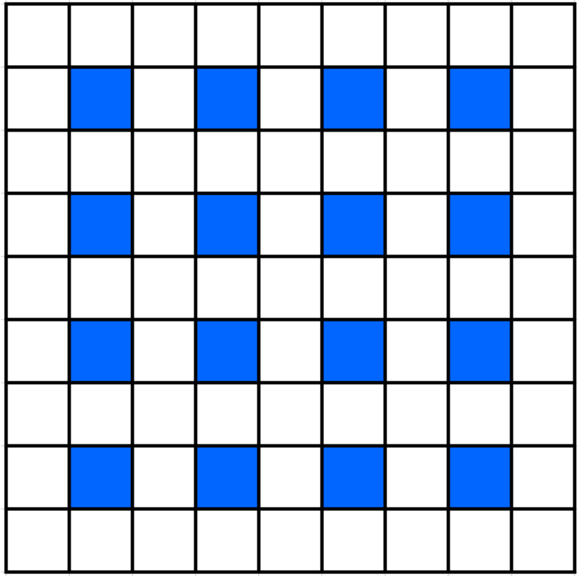
Approach



“**individually render lower resolution images corresponding to the regular grids**, and to then interleave the samples obtained this way by hand”

[Keller 2001]

Approach



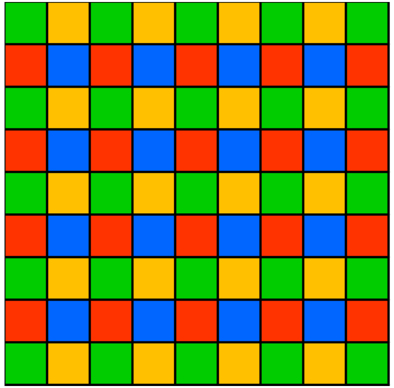
"**individually render lower resolution images corresponding to the regular grids**, and to then interleave the samples obtained this way by hand"

[Keller 2001]

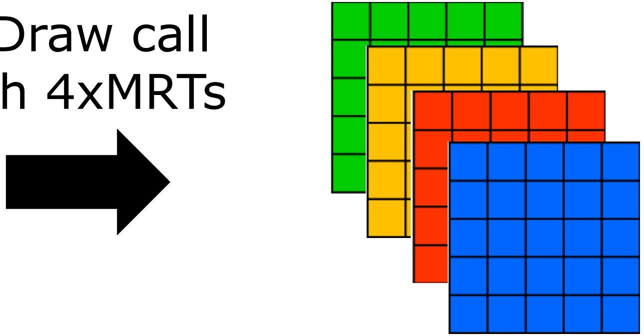
Our Solution:
"Interleaved Rendering"

Render each sampling pattern **separately**, using **downsampled** input textures

STEP 1: Deinterleave Input



1 Draw call
with 4xMRTs



Full-Resolution
Input Texture

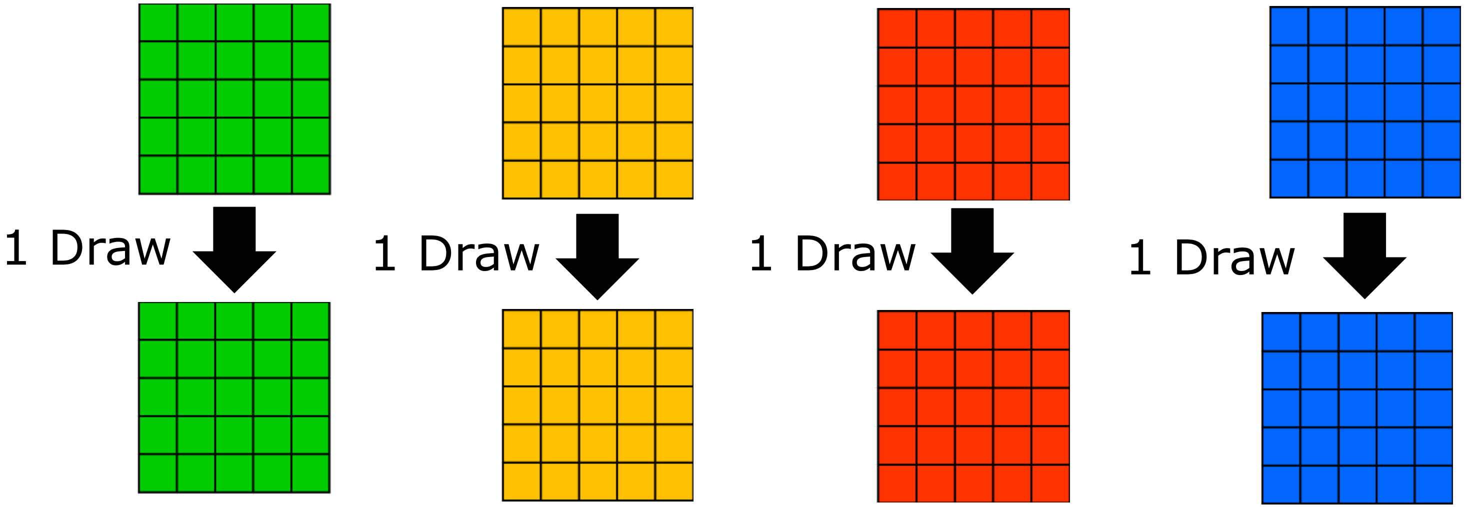
Width = W
Height = H

Half-Resolution
2D Texture Array

Width = iDivUp(W,2)
Height = iDivUp(H,2)

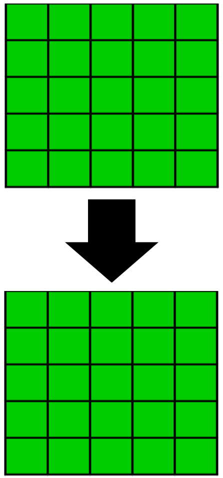
STEP 2: Jitter-Free Sampling

Input: Texture Array A (slices 0,1,2,3)



Output: Texture Array B (slices 0,1,2,3)

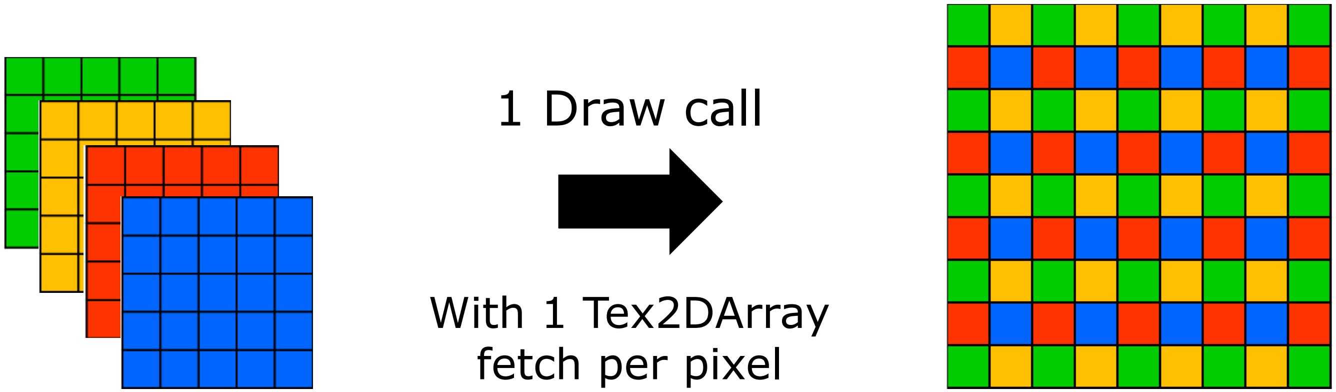
STEP 2: Jitter-Free Sampling



1. Constant jitter value per draw call
   ➔ better per-sample locality

2. Low-res input texture per draw call
   ➔ less memory bandwidth needed

STEP 3: Interleave Results

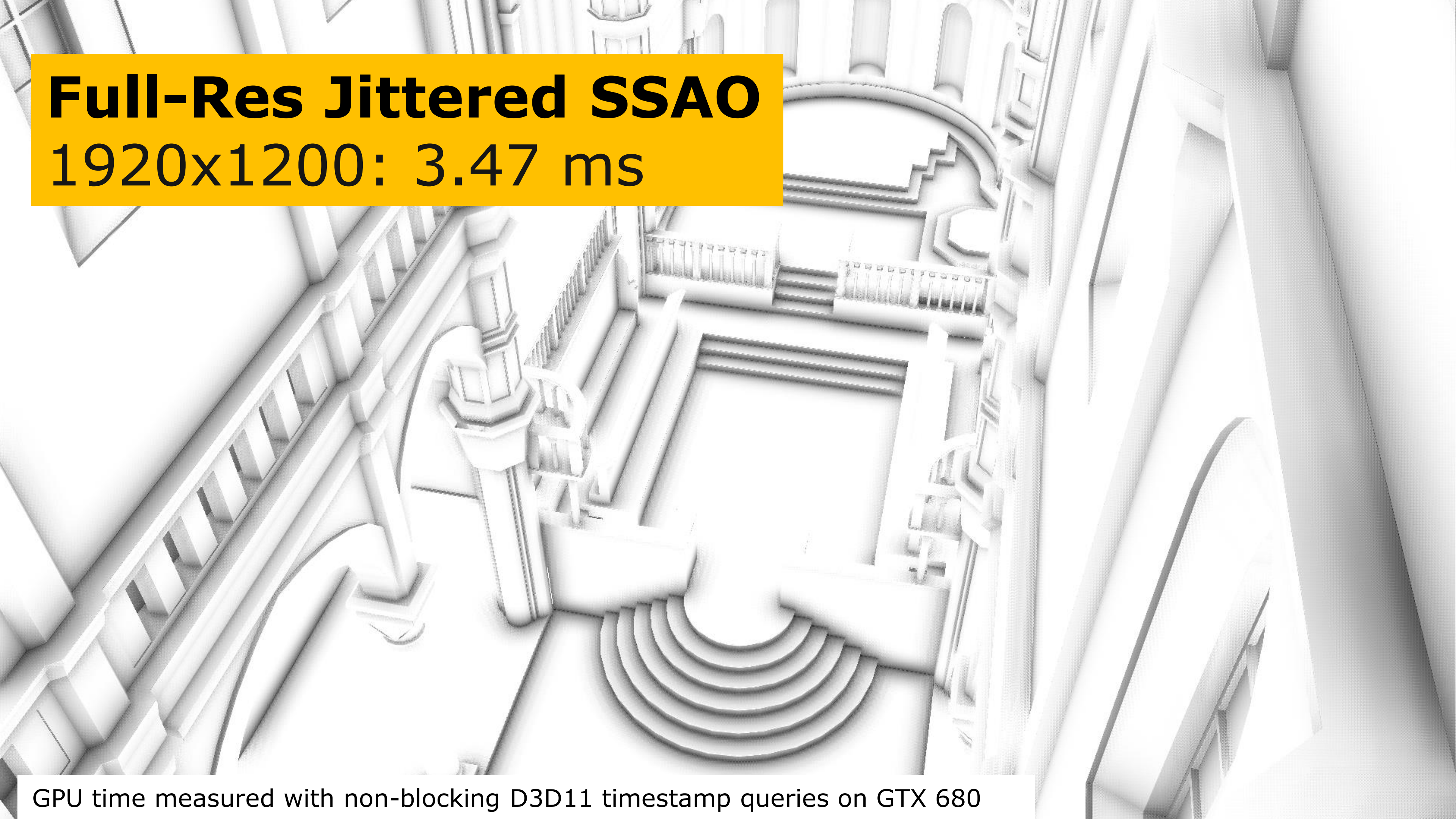


4x4 Interleaving

4x4 jitter textures are commonly used for jittering large sparse filters

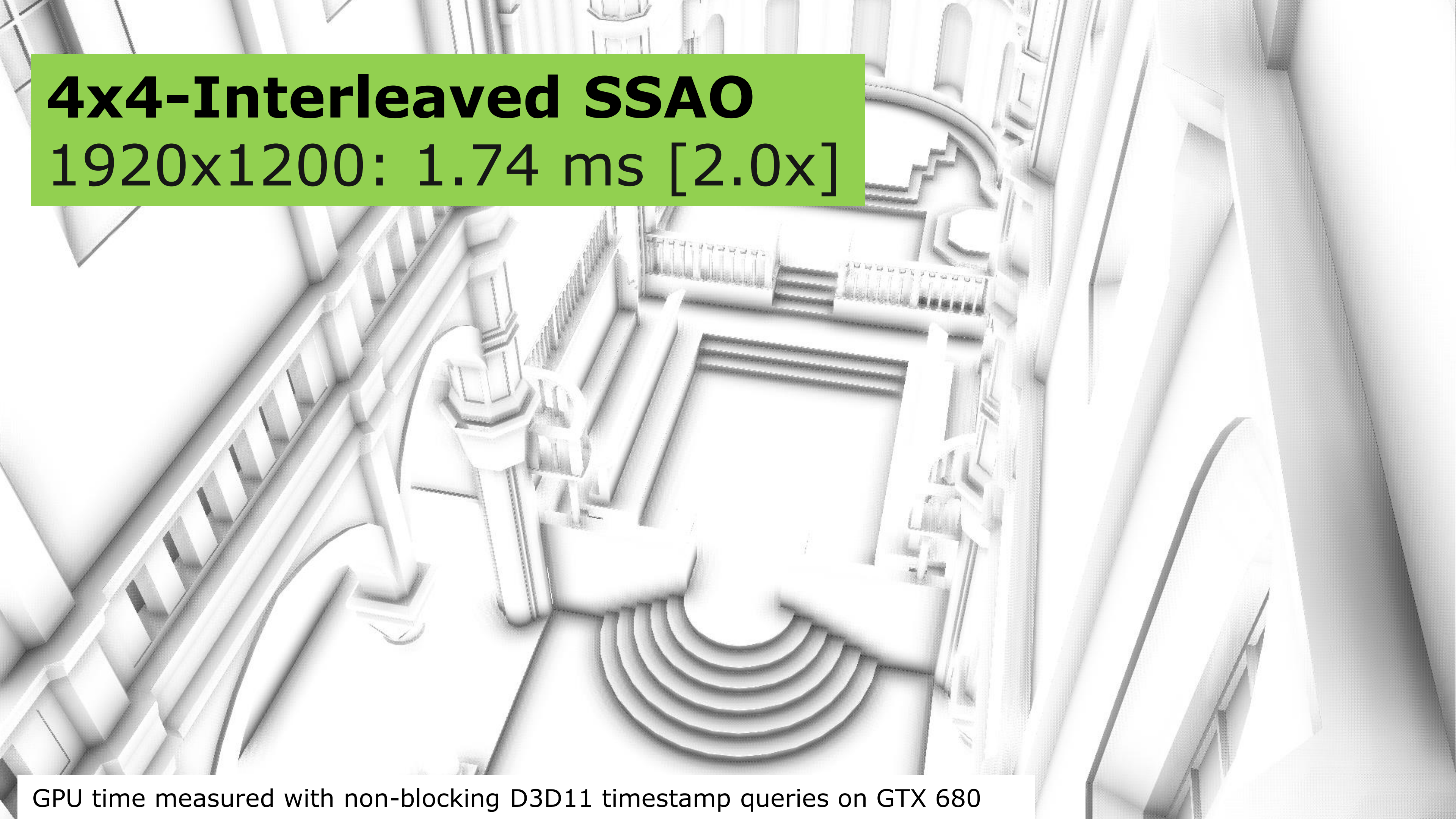
Can use a 4x4 interleaving pipeline

1. **Deinterleaving:** 2 Draw calls with 8xMRTs
2. **Sampling:** 16 Draw calls
3. **Interleaving:** 1 Draw call


Full-Res Jittered SSAO
1920x1200: 3.47 ms
GPU time measured with non-blocking D3D11 timestamp queries on GTX 680

4x4-Interleaved SSAO

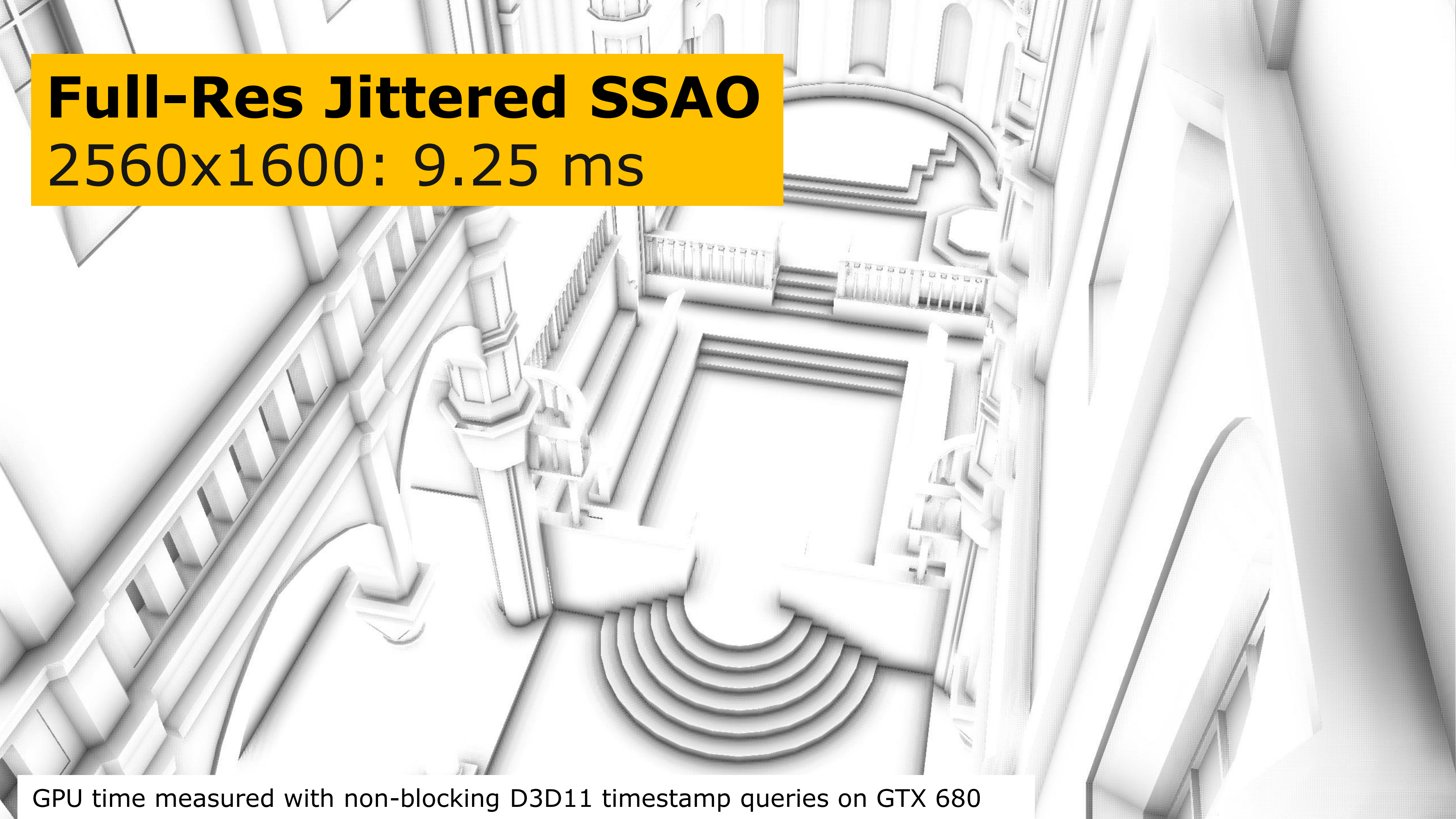
1920x1200: 1.74 ms [2.0x]



GPU time measured with non-blocking D3D11 timestamp queries on GTX 680

Full-Res Jittered SSAO

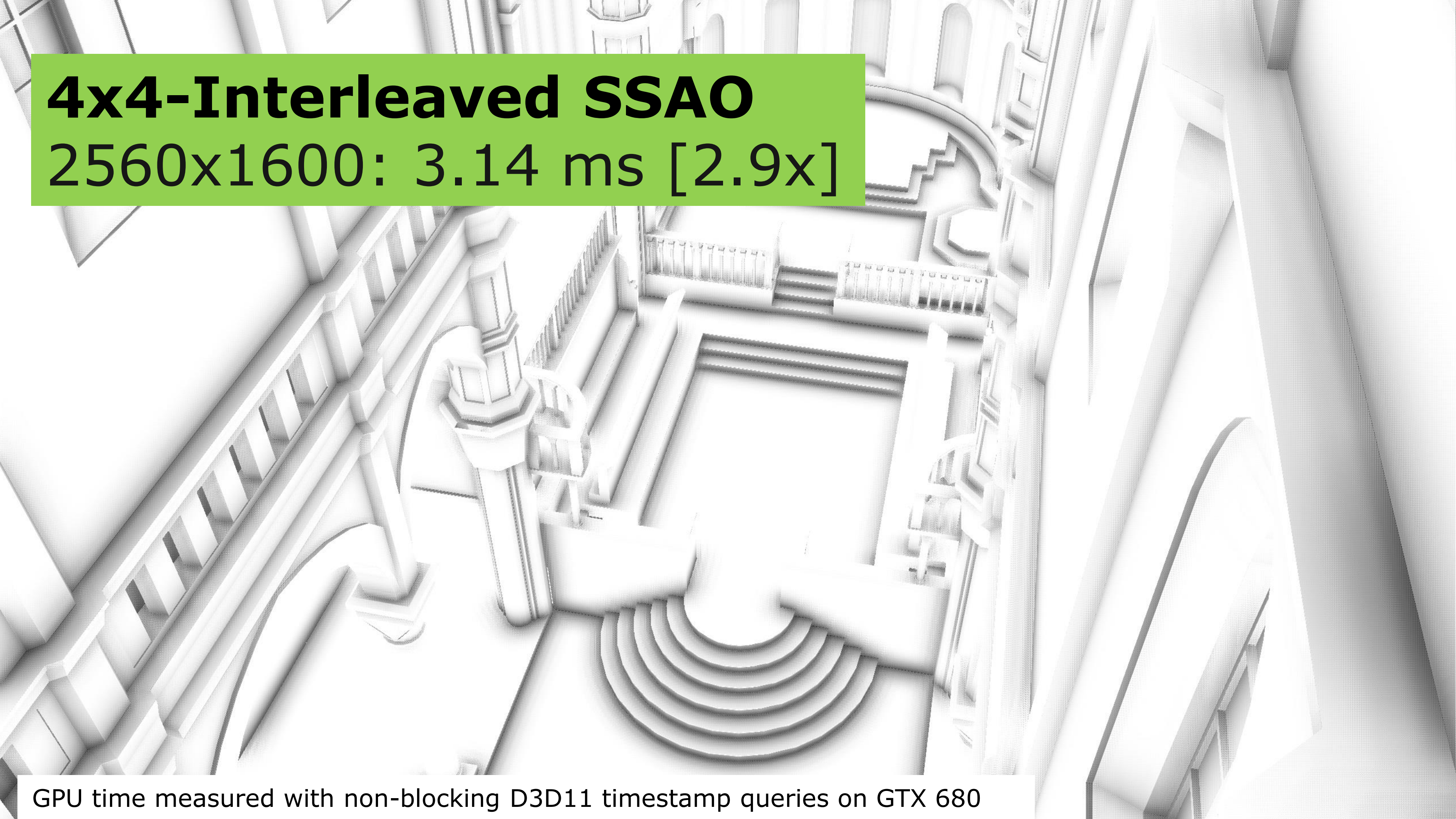
2560x1600: 9.25 ms



GPU time measured with non-blocking D3D11 timestamp queries on GTX 680

4x4-Interleaved SSAO

2560x1600: 3.14 ms [2.9x]



GPU time measured with non-blocking D3D11 timestamp queries on GTX 680

4x4-Interleaving Performance

GPU Times (in ms) *	1920x1200	2560x1600
STEP 1: Z Deinterleaving	0.12	0.21
STEP 2: SSAO	1.50	2.69
STEP 3: AO Interleaving	0.12	0.24
Total	**1.74**	**3.14**

* Measured with non-blocking D3D11 timestamp queries on GTX 680

Input = full-res R32F texture

Output = full-res SSAO

Texture-Cache Hit Rates

Can query per-draw cache texture-cache hit rates via:
NVIDIA PerfKit
AMD GPUPerfStudio 2

Example GPU counters *
tex0_cache_sector_misses
tex0_cache_sector_queries

1920x1200	GPU Time	Hit Rate
Non-Interleaved	3.47 ms	38%
4x4-Interleaved	1.50 ms	67%
Gain	2.3x	1.8x

* https://developer.nvidia.com/sites/default/files/akamai/tools/docs/PerfKit_User_Guide_2.2.0.12166.pdf

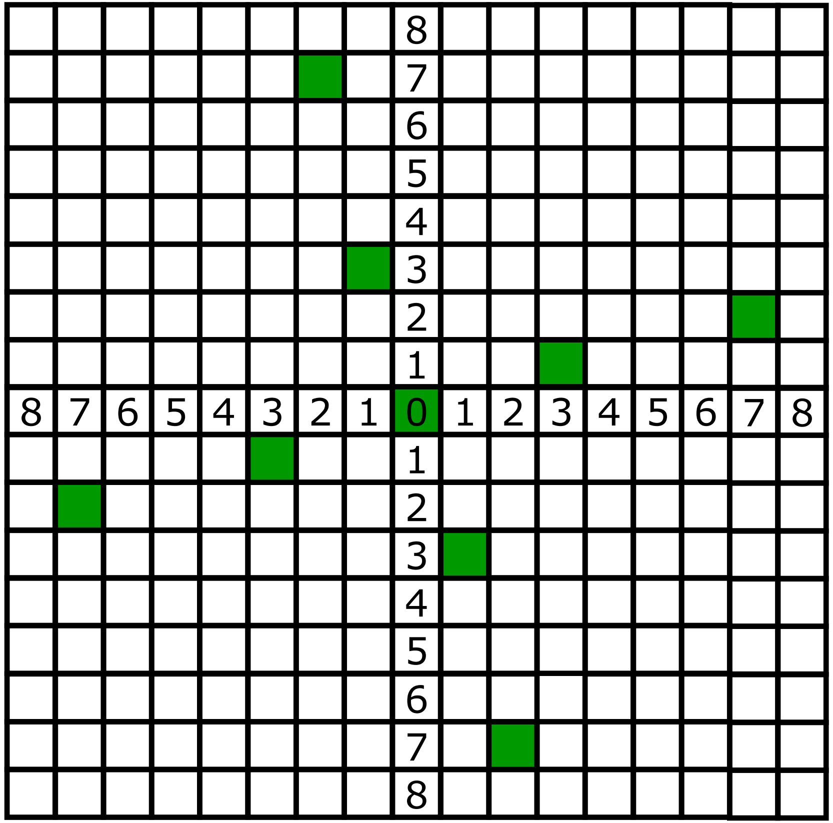
Texture-Cache Hit Rates

Can query per-draw cache texture-cache hit rates via:
NVIDIA PerfKit
AMD GPUPerfStudio 2

Example GPU counters *
tex0_cache_sector_misses
tex0_cache_sector_queries

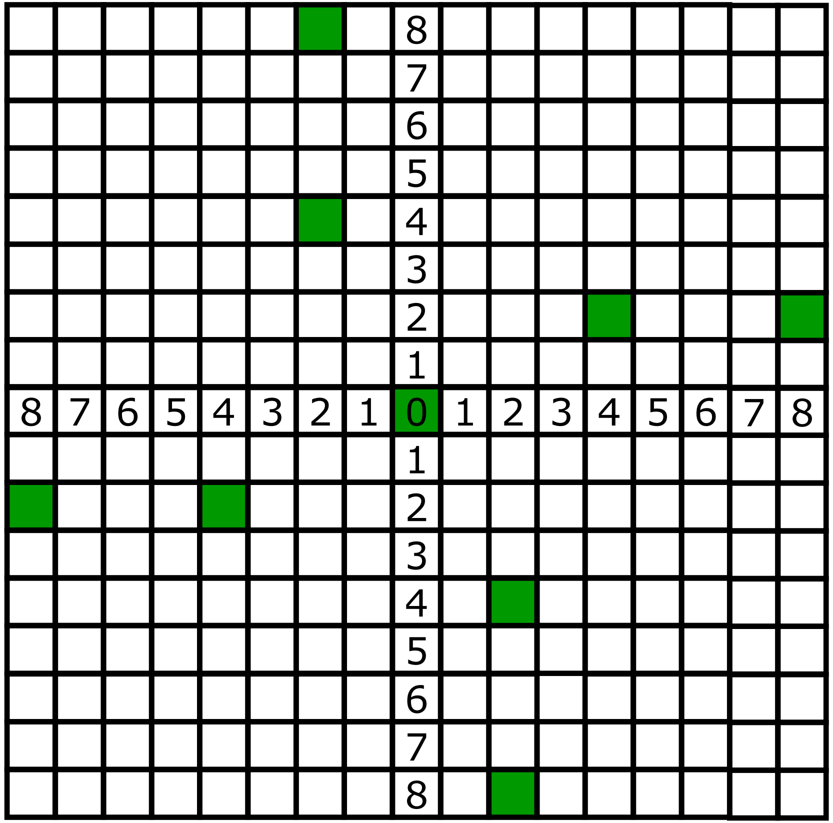
2560x1600	GPU Time	Hit Rate
Non-Interleaved	9.25 ms	32%
4x4-Interleaved	2.69 ms	62%
Gain	3.4x	1.9x

* https://developer.nvidia.com/sites/default/files/akamai/tools/docs/PerfKit_User_Guide_2.2.0.12166.pdf



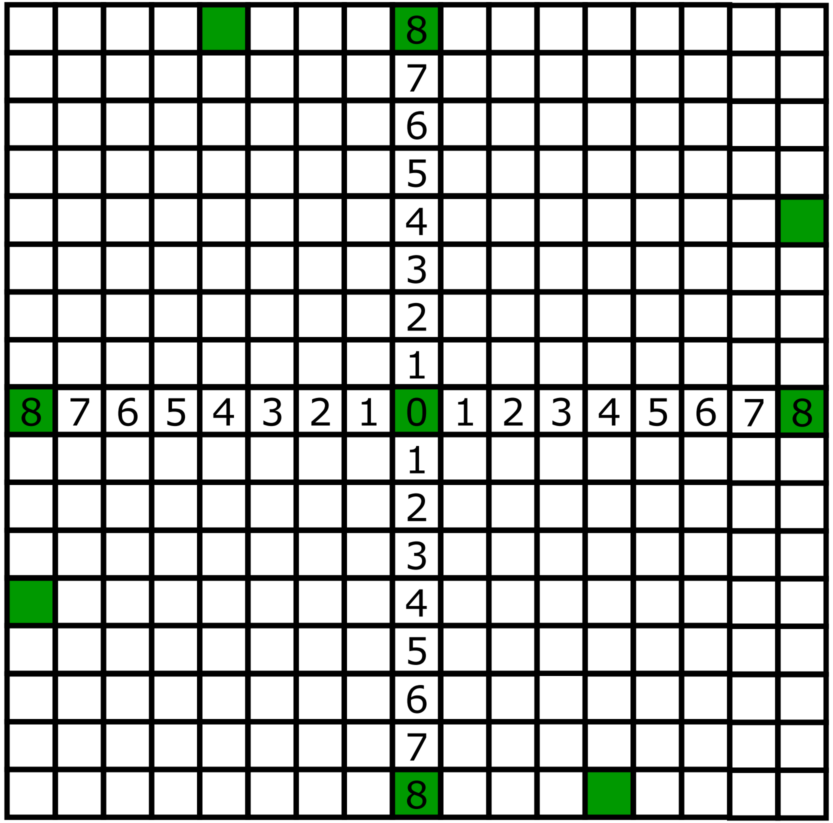
Example Sampling Pattern

With no Interleaved Rendering



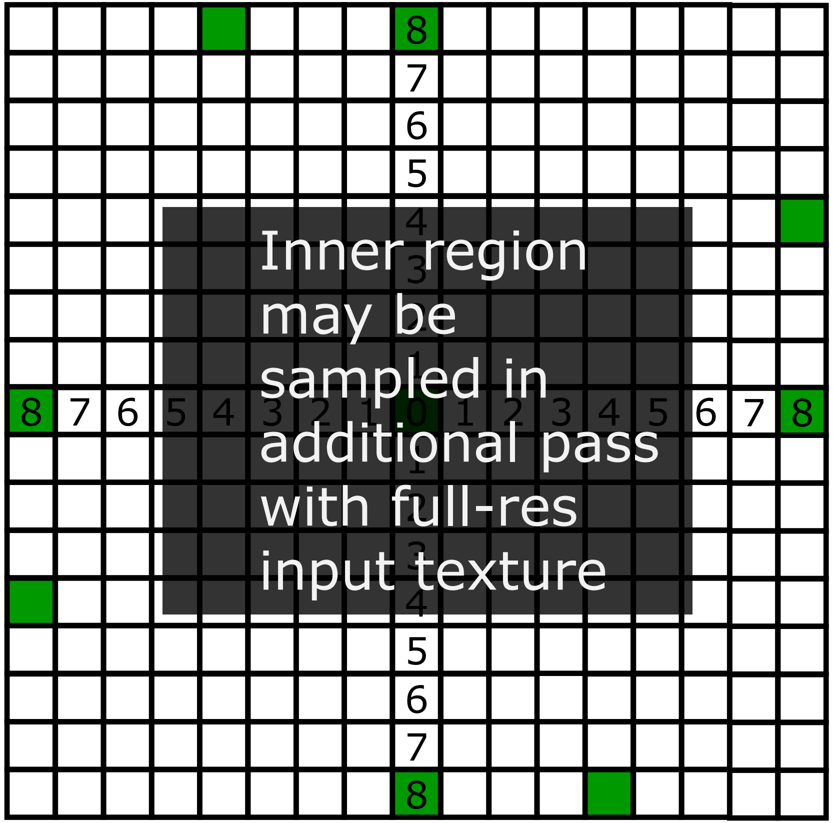
With 2x2 Interleaved Rendering

Sample coords are snapped to **half-res grid** aligned with kernel center



With 4x4 Interleaved Rendering

Sample coords are snapped to **quarter-res grid** aligned with kernel center



With 4x4 Interleaved Rendering

Sample coords are snapped to **quarter-res grid** aligned with kernel center

Interleaved Rendering: Wrap Up

- Improves performance
  - Better sampling locality
  - No jitter texture fetch anymore
- Looks the same
  - For large kernels (>16x16 full-res pixels)
  - Missed details for small kernels may be added back
- Used in shipping games
  - ArcheAge Online (2013)
  - The Secret World (2012)

4x4-Interleaved SSAO in Metro: Last Light (preview)

Acknowledgments

NVIDIA

- DevTech-Graphics
- Miguel Sainz
- Holger Gruen
- Yury Uralsky
- Alexander Kharlamov

Game Developers

- Funcom
- XL Games
- 4A Games
- DICE
- Crytek

Questions?

Louis Bavoil
lbavoil@nvidia.com

References

[Persson 2012] "Flexible Rendering for Multiple Platforms". Tobias Persson, Niklas Frykholm, BitSquid, 2012.

[McGuire 2012] "Scalable Ambient Obscurance". HPG 2012.
[Green 2012] "Volumetric Particle Shadows", NVIDIA Whitepaper. 2012.

[Gautron 2011] Pascal Gautron , Cyril Delalandre , Jean-Eudes Marvie, "Extinction transmittance maps". SIGGRAPH Asia 2011 Sketches.

[Jansen 2011] "Fast rendering of opacity mapped particles using DirectX 11 tessellation and mixed resolutions". Jon Jansen, Louis Bavoil. NVIDIA Whitepaper. 2011.

[Crytek 2011] Nickolay Kasyan, Nicolas Schulz, Tiago Sousa. "Secrets of CryENGINE 3 Graphics Technology". Advances in Real-Time Rendering Course. SIGGRAPH 2011.

[Jansen 2010] Jon Jansen and Louis Bavoil. "Fourier Opacity Mapping". I3D 2010.

[Salvi 2010] Marco Salvi, Kiril Vidimce, Andrew Lauritzen, and Aaron Lefohn, "Adaptive Volumetric Shadow Maps". Proceedings of EGSR 2010.

References

[Ritschel 2009] Tobias Ritschel, Thorsten Grosch, Hans-Peter Seidel. "Approximating Dynamic Global Illumination in Image Space". I3D 2009.

[Yuksel 2008] Cem Yuksel, John Keyser. "Deep Opacity Maps." Computer Graphics Forum (Proceedings of EUROGRAPHICS 2008).

[NVIDIA 2005] Hubert Nguyen and William Donnelly. "Real-time rendering and animation of realistic hair in 'Nalu'". In GPU Gems 2. 2005.

[Kniss 2003] Kniss, J., S. Premoze, C. Hansen, P. Shirley, and A. McPherson. 2003. "A Model for Volume Lighting and Modeling." IEEE Transactions on Visualization and Computer Graphics 9(2), pp. 150–162.

[Keller 2001] Alexander Keller and Wolfgang Heidrich. "Interleaved Sampling." Proceedings of the Eurographics Workshop on Rendering. 2001.

[Kim 2001] Tae-Yong Kim and Ulrich Neumann. "Opacity Shadow Maps". Proceedings of the 12th Eurographics Workshop on Rendering Techniques. 2001.

[Lokovic 2000] Tom Lokovic, Eric Veach. "Deep Shadow Maps". SIGGRAPH 2000.